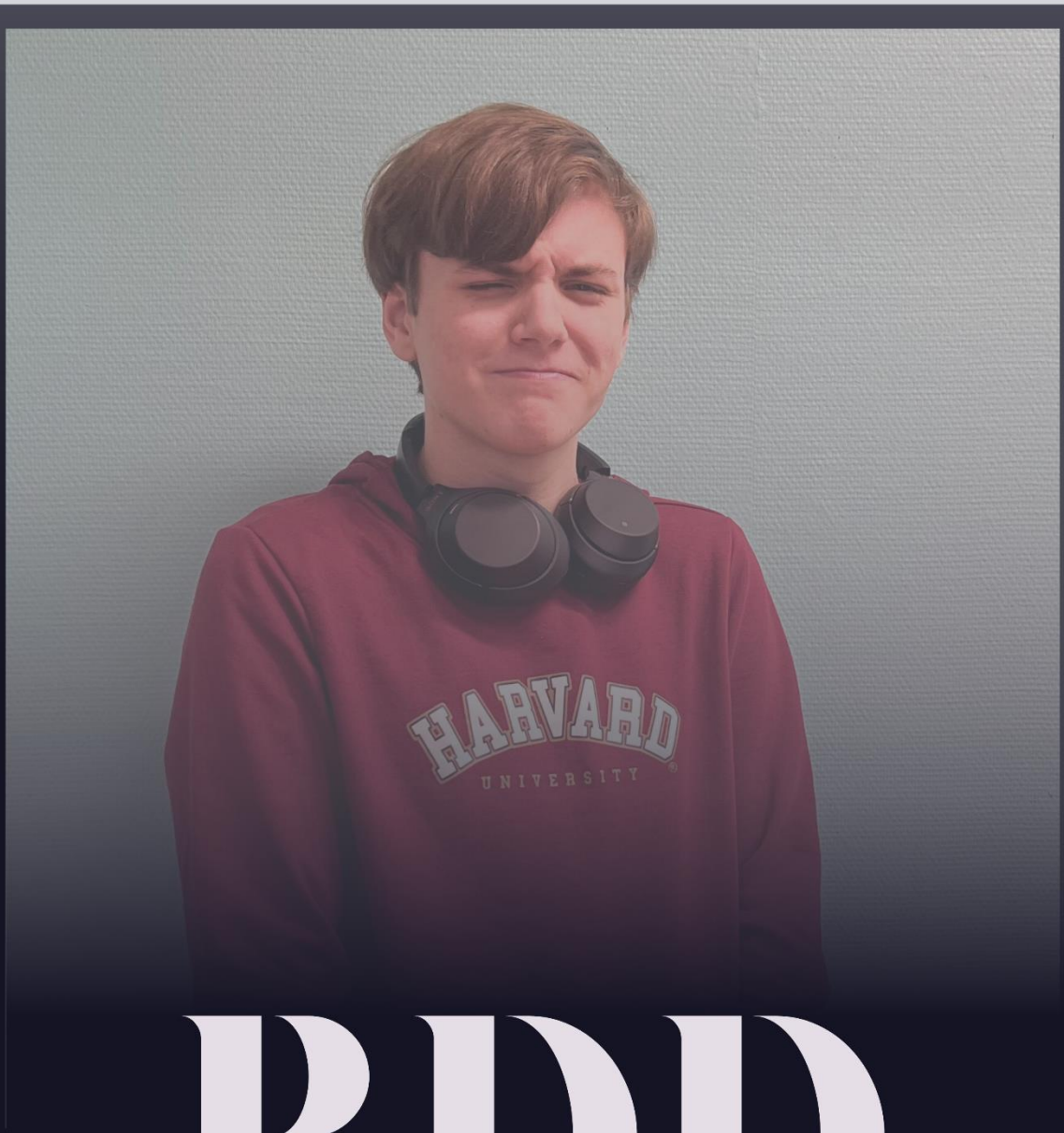


# BDD

## Base De Données - SQL c'est le Système Qui Lag

---

Par William Cloudless

# Table des matières

# Le langage SQL

Le SQL signifie « Structured Query Language ». Il contient :

- DDL : Un langage de description
- DML : Un langage de manipulation de données
- DCL : un langage de gestion de protections

| DDL | DML | DCL | Embedded |
|---|---|---|---|
| ALTER | DELETE | GRANT | EXEC-SQL |
| CREATE | INSERT | REVOKE | END-EXEC |
| DESCRIBE | SELECT | | DECLARE CURSOR |
| DROP | UPDATE | | SELECT INTO |
| RENAME | TRUNCATE | | SQLCODE |

## Manipulation de données (DML)

### Les commandes

#### SELECT

Il s'agit d'une commande permettant de récupérer les données d'une table.

*Syntaxe :*

```
SELECT [colonne]
FROM [table];
```

Lors de l'exécution de la commande, le résultat (s'il y en a) peut contenir des doublons. Pour éviter cela, il faut ajouter la mention « DISTINCT ».

*Syntaxe :*

```
SELECT DISTINCT [colonne]
FROM [table];
```

#### WHERE

Il s'agit d'une commande permettant de restreindre les données ne remplissant pas la condition renseignée. (Similaire au « *if* » en java)

*Syntaxe :*

```
SELECT [colonne] FROM [table]
WHERE [condition];
```

Les conditions peuvent contenir les qualifications suivantes >, >=, <, <=, =, <> et **AND**, **OR**, **NOT**.

Exemple de conditions :

```
SELECT * FROM produits
WHERE poids > 15;

SELECT * FROM produits
WHERE couleur='rouge' AND poids > 30;
```

Des raccourcis d'écritures existent et permettent d'écrire plus rapidement des commandes SQL. Il y a la qualification **IN** et **BETWEEN**.

Exemple de commande :

```
SELECT * FROM fournisseurs
WHERE ville IN ('Lille','Lyon','Nice');

SELECT * FROM produits
WHERE poids BETWEEN 15 AND 35;
```

Des recherches peuvent être faites sur les colonnes avec les opérateurs **LIKE** et **SIMILAR TO**.

Pour l'opérateur **LIKE**, il permet de faire des recherches approximatives dans une chaîne de caractères. Les « jokers » sont :

- % pour remplacer entre zéro à plusieurs lettres
- _ pour une exactement lettre

Exemple de commande :

```
SELECT * FROM fournisseurs
WHERE nom LIKE 'D%'
```

Cette commande renvoie tous les fournisseurs dont le nom commence par un « D ».

Les opérateurs >, < ,= , <>, **BETWEEN**, **IN**, **DISTINCT** fonctionnent :

- Sur des chaînes
- Sur des entiers
- Sur des réels
- Sur des dates

Une case non remplie est annotée à la méta-valeur **NULL**.

Les prédicats **IS NULL** ou **IS NOT NULL** permettent de la tester. Exemple de commandes :

```
SELECT * FROM fournisseurs
WHERE adresse IS NULL;
```

Cette commande renvoie les fournisseurs qui n'ont pas d'adresse renseignée.

```
SELECT * FROM fournisseurs
WHERE adresse IS NOT NULL ;
```

Cette commande renvoie les fournisseurs qui ont une adresse renseignée.

## ORDER BY

Cette clause permet de trier sur une ou plusieurs colonnes de façon croissante ou décroissante.

```
SELECT design, couleur FROM produit
WHERE couleur IN ('rouge','vert','bleu')
ORDER BY design ASC, couleur DESC;
```

Cette commande renvoie le design et la couleur si le produit est rouge ou vert ou bleu trié sur le nom de manière croissante et la couleur de manière décroissante.

## INSERT

Cette commande permet d'insérer des données dans une table. Il y a deux formes possibles :

### *Insertion en extension (**VALUES**)*

Pour insérer une ligne, on utilise cette syntaxe :

```
INSERT INTO [table] ([colonnes])
VALUES ([valeurs]);
```

On précise les colonnes qu'on souhaite modifier. Ceux qui n'ont pas été mentionnés seront mis en **NULL**.

Dans le cas où on ne préciserait pas les colonnes, alors toutes sont concernées.

### *Insertion en intention (**SELECT**)*

Pour insérer une ou plusieurs lignes depuis une autre table, on utilise cette syntaxe :

```
INSERT INTO [table]
SELECT [colonnes] FROM [table]
WHERE [condition]
```

Exemple :

```
INSERT INTO braderie(pno,prix)
       SELECT pno, 100 FROM produits
       WHERE produit.prix < 500 AND produit.prix > 100;
```

Cette commande permet d'ajouter dans la table « braderie » les numéros de produits entre 100€ et 500€ et les afficher à un prix de 100€

## UPDATE

Il permet de modifier une ou plusieurs colonnes sur des lignes concernée par une condition. On utilise cette syntaxe :

```
UPDATE [table]
SET [colonne]=[valeur]
WHERE [condition];
```

Cette commande modifie plusieurs lignes simultanément !

Exemple de commande :

```
UPDATE produits
SET prix = prix*1.05
WHERE couleur='rouge';
```

Il permet d'augmenter le prix de 5% pour tous les produits rouges.

On peut aussi utiliser des sous-requêtes. Exemple :

```
UPDATE commandes
SET qute = (SELECT qute FROM modifs
            WHERE commandes.cno = modifs.cno)
WHERE commandes.cno IN (SELECT cno FROM modifs);
```

Cette commande modifie les quantités en commandes avec les nouvelles quantités issues d'une autre table.

Le SQL permet aussi de faire des calculs. Exemple :

```
UPDATE TABLE [table]
SET cle=cle+1 ;
```

Incrémente de 1 toutes les clés.

### DELETE

Il permet de supprimer les lignes concernées par la condition indiquée. On utilise cette syntaxe :

```
DELETE FROM [table] WHERE [condition];
```

On peut supprimer les données dans la table complète si on ne précise pas de condition (`DELETE FROM [table]`). La table reste existante, mais elle est vide.

On peut aussi utiliser des sous-requêtes. Exemple :

```
DELETE FROM produits
WHERE pno IN (SELECT pno
              FROM commandes
              GROUP BY pno
              HAVING SUM(qute)<10);
```

Supprime les produits qui n'ont pas été commandés en un minimum 10 exemplaires au total.

### TRUNCATE

Il permet de supprimer toutes les données d'une table. On utilise cette syntaxe :

```
TRUNCATE [table] [RESTART IDENTITY]
```

L'option « `RESTART IDENTITY` » permet de réinitialiser les numéros automatiques.

## Jointure

### SQL-89

Pour associer plusieurs tables ensemble, on peut faire cette commande :

```
SELECT * FROM fournisseurs, produits;
```

Cette commande liste le produit cartésien de la table « fournisseurs » et « produits ».

Cependant, si on souhaite faire une équi-jointure, on doit indiquer une restriction avec **WHERE**.

```
SELECT nom, pno, qute
FROM fournisseurs, commandes
WHERE fournisseurs.fno = commandes.fno;
```

Cette commande associe donc les fournisseurs par commande.

### SQL-92

Pour associer plusieurs tables ensemble, on utilise « **INNER JOIN** ».

Exemple de commande :

```
SELECT nom, pno, qute
FROM fournisseur f INNER JOIN commande C
     ON f.fno=C.fno;
```

Cette commande associe donc les fournisseurs par commande.

De cette convention, sont apparues plusieurs commandes qui permettent d'effectuer d'autre jointures externes :

- **LEFT OUTER JOIN** (Plus d'information : sql.sh)
- **RIGHT OUTER JOIN** (Plus d'information : sql.sh)
- **FULL OUTER JOIN** (Plus d'information : sql.sh)
- **CROSS JOIN** (Plus d'information : sql.sh)

## Utilisation de synonyme de noms de tables

Lors de très grosse commande SQL, il peut être intéressant d'appeler une table avec un nom plus simple et plus court. Il est nécessaire d'utiliser ces synonymes si un vous faites une jointure d'une table avec la même table. C'est en utilisant « **AS** » que l'on définit un synonyme d'une table.

Exemple de commande :

```
SELECT fournisseur.nom, commandes.pno, commandes.qute
FROM fournisseurs, commandes
WHERE fournisseurs.fno = commandes.fno;
```

Cette commande liste les noms de fournisseurs avec les numéros de produits commandés ainsi que la quantité commandée.

```
SELECT f.nom, C.pno, C.qute
FROM fournisseurs AS f, commandes AS C
WHERE f.fno = C.fno ;
```

Cette commande affiche le même résultat que la commande précédente, mais elle utilise des synonymes.

## Effectuer des calculs

Règles syntaxiques :

- Pas de colonnes dans une clause HAVING
- Pas de fonction d'agrégat dans une clause WHERE
- Toutes les colonnes du SELECT doivent être dans le GROUP BY (et l'inverse n'est pas vrai)

## Calcul sur les lignes

SQL permet de faire des calculs sur les colonnes. Exemple :

```
SELECT pno, design, prix*1.206 AS prixTTC
FROM produits;
```

Cette commande affiche un produit et son prix en TTC par ligne.

D'autre fonctions existent :

| Logiques | Maths | Chaînes | Dates |
|---|---|---|---|
| and | + - * / % | \|\| | + - |
| or | abs | length | current_date |
| not | ceil | lower | current_time |
| | floor | upper | age |
| | round | substring | extract |
| | mod | trim | to_char |
| | power | repeat | |
| | random | replace | |

## Calcul sur les colonnes

Calculs effectués sur des regroupements d'un ensemble de tuples de la table

| | |
|---|---|
| AVG | Moyenne |
| SUM | Somme |
| MAX | Maximum |
| MIN | Minimum |
| COUNT | Nombre d'éléments |

Exemple de commande :

```
SELECT SUM(qute)
FROM commandes;
```

Renvoie la somme des commandes.

```
SELECT COUNT(*) AS nbre
FROM commandes
WHERE pno=102;
```

Renvoie le nombre de commandes du produit n°102.

Sémantique du COUNT :

- COUNT(*) toutes les lignes
- COUNT(fno) les valeurs non nulles
- COUNT(DISTINCT fno) les valeurs distinctes non nulles

### Regroupement

On utilise désormais « **GROUP BY** » pour faire des opérations statistiques sur des groupes de données ayant une caractéristique commune. Exemple :

```
SELECT fno, SUM(qute)
FROM commandes
GROUP BY fno;
```

Affiche la somme des quantités commandées à chaque fournisseur.

#### *Test sur les valeurs regroupées*

Pour mettre des conditions sur les fonctions de regroupement, on utilise la clause « **HAVING** ». Exemple :

```
SELECT fno, COUNT(*)
FROM commandes
GROUP BY fno
HAVING COUNT(*)<3 ;
```

Liste les fournisseurs qui ont moins de 3 commandes.

Des opérations de regroupements plus complexes sont possibles en utilisant la notion d'ensembles de regroupement

- **GROUP BY GROUPING SET**((c1,c2),(c3,c4),(),...)
  Regroupe sur les valeurs citées
- **GROUP BY ROLLUP**(c1,c2,c3,...)
  Regroupe sur tous les préfixes des colonnes citées
- **GROUP BY CUBE**(c1,c2,c3,...)
  Regroupe sur tous les sous-ensembles possibles

**ROLLUP**(x,(y, z),t) est équivalent à **GROUPING SETS**((x,y,z,t),(x,y,z),(x),())

## Sous-requête

### Sous-requête à valeur unique

SQL donne la possibilité d'utiliser des sous-requêtes afin de décrire des requêtes complexes permettant d'effectuer des opérations dépendant d'autres requêtes. Exemple :

```
SELECT nom
FROM fournisseurs
WHERE ville = (SELECT ville
               FROM fournisseurs
               WHERE fno=10);
```

Cette requête renvoie les fournisseurs qui habitent à la même ville que le fournisseur 10.

### Sous-requête renvoyant un ensemble

Pour permettre la comparaison avec un ensemble, des opérateurs spéciaux existent :

- **IN**
- > **ALL**
- > **ANY**

Exemple :

```
SELECT *
FROM produits
WHERE pno IN (SELECT pno FROM commandes);
```

Permet de lister les informations sur les produits en commande.

Ces deux commandes sont équivalentes :

| | |
|---|---|
| `SELECT *`<br>`FROM table1`<br>`WHERE champ1`<br>`IN (SELECT champ1 FROM table2);` | `SELECT DISTINCT table1.*`<br>`FROM table1`<br>`INNER JOIN table2`<br>`ON table1.champ1=table2.champ1;` |

Ces deux commandes sont également équivalentes :

| | |
|---|---|
| `SELECT * FROM table1`<br>`WHERE champ1 NOT IN (SELECT`<br>`champ1 FROM table2);` | `SELECT table1.* FROM table1`<br>`LEFT JOIN table2 ON`<br>`table1.champ1=table2.champ1`<br>`WHERE table2.champ1 IS NULL;` |

Il existe aussi le test d'existence « EXISTS » qui permet de tester si dans un ensemble, il existe les tuples renseignées. Exemple :

```
SELECT DISTINCT c1.fno FROM commandes c1
WHERE NOT EXISTS (SELECT * FROM commandes c2
                  WHERE c2.fno = 19 AND c2.qute >= c1.qute);
```

Permet de lister les fournisseurs d'au moins un des produits fournis aussi par un fournisseur d'un produit rouge.

### Les opérations ensemblistes

Ce sont des opérateurs qui s'intercalent entre deux commande SELECT. Les opérateurs sont :

- **UNION**: fournis la réunion des tuples des 2 **SELECT**.
- **INTERSECT**: fournis l'intersection des tuples des 2 **SELECT**.
- **EXCEPT**: fournis les tuples du 1$^{er}$ **SELECT** qui ne sont pas dans le second.

```
SELECT pno FROM produits WHERE poids>20
UNION ALL
SELECT pno FROM commandes WHERE fno=15;
```

Renvoie les numéros de produits dont le poids est supérieur à 20 ainsi que les produits commandés par le fournisseur 15.

Remarque :

- `INTERSECT` et `EXCEPT` sont rarement implémentés.
- `UNION` supprime par défaut les doublons de l'opération. Si les doublons sont souhaités, il faut utiliser la forme `UNION ALL`.

# Data Description Language (DDL)

Les types sont dépendants des SGBD. Ils peuvent changer de MySQL à PostgreSQL.

## Les types de données

### Type alphanumérique

- `CHAR(n)` : longueur fixe de n caractères (max : 16383)
- `VARCHAR(n)` : longueur variable de n caractères max.
- `TEXT`: longueur variable non bornée (=`CLOB`)

### Types numériques

- `NUMERIC(n,[d])` : nombre de n chiffres dont d après la virgule.
- `SMALLINT` : nombre sur 2 octets
- `INT` : nombre sur 4 octets
- `BIGINT` : nombre sur 8 octets
- `FLOAT` : numérique flottant sur 8 octets

`NUMERIC` s'écrit parfois `NUMBER` ou `DECIMAL`.

### Type gestion du temps

- `DATE` : exemple -> 23/03/2024
- `TIME` : exemple -> 14:45:10:95 (avec ou sans Timezone)
- `TIMESTAMP` : regroupe `DATE` et `TIME` (avec ou sans Timezone)
- `INTERVAL` : durée entre 2 `DATE` ou `TIME`.

Les `INTERVAL` peuvent s'exprimer de nombreuses manières. Chaque SGBD a sa propre valeur par défaut (en postgres nbjours pour les dates, et hh:mm:ss pour les time)

### Types binaires

- `BIT(n)` : s'écrit B'10111'
- `VARBIT(n)` :
- `BOOLEAN` : `TRUE` ou `FALSE`
- `BLOB` : binary large object

### Conversion entre types

Pour convertir un type, on utilise la fonction `CAST` ou la notation « `::` ». Exemple :

```
SELECT '2023-01-25' + 1 ;
```

Renvoie une erreur : `ERROR invalid input syntax for type integer`

```
SELECT '2023-01-25'::DATE + 1;
```

Renvoie: `2023-01-26`

```
SELECT '2023-01-25'::DATE + '1 month'::interval ;
```

Renvoie: `2023-02-25`

## Les commandes

### CREATE

Pour créer un objet et plus généralement une table, on utilise cette syntaxe :

```
CREATE TABLE [table]
(
        [nom variable] : [type],
        [nom variable] : [type],
        [contrainte de table]
)
```

Dans la plupart des SGBD, le nom de la table doit commencer par une lettre, 254 colonnes maximum par table.

On peut aussi créer une table avec une commande **`SELECT`** ce qui donnera à la table créer le schéma de la précédente requête. Exemple :

```
CREATE TABLE Bonus AS
      SELECT nom, salaire FROM Employe
      WHERE métier = 'Chef de service';
```

Attention ! Les contraintes de la table effectuées par la requête ne sont pas copiées sur la nouvelle table.

*Les contraintes de colonne*

Les contraintes spécifient des obligations sur des colonnes. Il y a :

- `NOT NULL` : force la saisie de la colonne (ne peut pas être `NULL`)
- `UNIQUE` : vérifie que toutes les valeurs de la colonne sont différentes.
- `CHECK([condition])` : vérifie la condition à chaque mise à jour.
- `DEFAULT [valeur]` : précise une valeur par défaut.
- `CONSTRAINT [nom]` : permet de nommer une contrainte.

Exemple :

```
CREATE TABLE fournisseurs
(
      fno INTEGER NOT NULL
      CONSTRAINT c1 CHECK (fno > 100),
      nom CHAR(20) UNIQUE NOT NULL,
      adresse CHAR(50),
      ville CHAR(10)
      DEFAULT 'Lille'
);
```

*Les contraintes de table*

Elles s'appliquent en général sur plusieurs colonnes.

- **`CONSTRAINT`** `nom` : permet de nommer une contrainte
- **`PRIMARY KEY`** `[colonne]` : définition d'une clé primaire
- **`FOREIGN KEY`** `[colonne]` : définition d'une clé étrangère

Exemple :

```
CREATE TABLE commandes
(
     fno INTEGER,
     pno INTEGER,
     quté INTEGER,
     CONSTRAINT pk_commandes PRIMARY KEY(fno,pno),
     CONSTRAINT fk_fournisseur FOREIGN KEY(fno)
     REFERENCES fournisseurs(fno),
     CONSTRAINT fk_produits FOREIGN KEY(pno)
     REFERENCES produits(pno)
);
```

Comme sur Access, on peut définir les actions qui se répercuteront sur les autres tables contenant une ou plusieurs clés étrangères. On utilise :

- Quand on supprime : **`ON DELETE`** {**`RESTRICT`** | **`CASCADE`** | **`SET NULL`** | **`SET DEFAULT`**}
- Quand on modifie : **`ON UPDATE`** {**`RESTRICT`** | **`CASCADE`** | **`SET NULL`** | **`SET DEFAULT`**}

**`RESTRICT`** est mis par défaut quand rien n'est renseigné.

Exemple :

```
CREATE TABLE Ici
(
    a INTEGER,
    b INTEGER,
    CONSTRAINT pk_ici PRIMARY KEY(a),
    CONSTRAINT fk_labas FOREIGN KEY(b)
                        REFERENCES Labas(b)
                        ON UPDATE CASCADE
);
```

### DESCRIBE

Cette commande permet d'afficher la structure d'une table. Exemple :

```
DESCRIBE fournisseurs;
```

Cette commande affiche la structure de la table Fournisseurs.

### ALTER

Cette commande permet de modifier la structure d'une table. On peut ajouter, modifier et supprimer une colonne. La syntaxe est :

```
ALTER TABLE [table]
      (ADD | ALTER | DROP | RENAME) [value];
```

Exemple de commande:

```
ALTER TABLE fournisseurs
      ADD prenom CHAR(15);
```

Ajoute la colonne « prenom ».

```
ALTER TABLE commandes
      MODIFY quté NUMERIC(8,2);
```

Modifie la colonne « quté » avec 8 chiffres et 2 après la virgule.

```
ALTER TABLE commandes
      DROP quté;
```

Supprime la colonne « quté ».

### DROP

Cette commande permet de supprimer une table. La syntaxe est :

```
DROP TABLE [ IF EXISTS ] [table] [CASCADE | RESTRICT]
```

Des paramètres supplémentaires peuvent être mis en plus de **DROP TABLE** [table] :

- **IF EXISTS** : Supprime la table s'il elle existe.
- **CASCACE**: Supprime tous ce qu'il y a un rapport avec la table (les vues, les clés, etc…)
- **RESTRICT** : Oblige la vérification des valeurs référencées par une clé étrangère. S'il y en a, alors l'action est annulée.

## Les vues

Une vie est une « fenêtre » sur la base de données permettant à chacun de voir les données comme il le souhaite. Il faut utiliser le **SELECT** pour sélectionner les colonnes. On utilise cette syntaxe pour définir une vue :

```
CREATE VIEW <nom-de-vue>
       (<nom-de-colonne>, ...)
       AS <sélection>;
```

On peut modifier une vue avec **ALTER** et supprimer une vue avec **DROP**.

ATTENTION ! Une vie ne contient pas de données. Les données sont stockées dans les tables d'origine. Les vues marchent comme une table, on peut utiliser la commande **SELECT** dessus.

Exemple de commande :

```
CREATE VIEW vuefournisseur
AS SELECT f.nom, f.adresse, SUM(C.qté) AS somme
   FROM fournisseurs f, commandes C
   WHERE f.fno = C.fno
   GROUP BY f.nom, f.adresse;
```

Cette commande créée une vue sur la jointure fournisseurs-commandes ne permettant de voir que les fournisseurs avec leur total de commandes.

### Avantages

- Une vue permet de restreindre l'accès d'une table à un sous-ensemble de colonnes et un sous-ensemble de lignes.
- Les vues permettent de rendre les programmes moins dépendants de l'évolution des tables.
- Il est possible de rassembler dans un seul objet des données qui sont dans plusieurs tables.
- La programmation est simplifiée pour l'utilisateur puisqu'une partie de l'ordre **SELECT** est déjà codée dans la vue.
- Les vues permettent de simplifier les ordres **SELECT** avec sous-requêtes complexes (une vue par sous-requête).

### Inconvénients

Si vous souhaitez mettre à jour des données via une vue, il faut créer une « règle » avec **RULE**. On utilise cette syntaxe :

```
CREATE [ OR REPLACE ] RULE name AS ON event
         TO TABLE_NAME [ WHERE condition ]
         DO [ ALSO | INSTEAD ] { NOTHING | command | ( command
command ... ) };
```

Exemple de commande :

```
CREATE RULE r1 AS ON UPDATE TO vuefournisseur
DO INSTEAD INSERT INTO commandes VALUES (NEW.nom,NEW.somme);
```

Attention, les mises à jour sont autorisées si et seulement si :

- La clause **FROM** principale ne fait référence qu'à une seule table ou une vue accessible en mise à jour.
- Elle ne comporte pas de **DISTINCT** ou une fonction sur colonne.
- Elle ne contient pas de clause **GROUP BY** ou **HAVING**.
- Elle n'utilise pas les opérateurs ensemblistes **UNION**, **INTERSECT** ou **EXCEPT**
- Elle ne contient que des références aux colonnes de la table source (pas de **COUNT**, **SUM**, . . .).
- Elle ne contient pas de sous-requête dont la clause **FROM** contient la même table que la clause **FROM** principale.

## Data Control Language (DCL)

Différentes personnes peuvent travailler sur la même base de données. C'est pour cela que l'on doit attribuer des droits par utilisateur.

Au départ, il existe le DBA (DataBase Administrator). Il s'agit du super utilisateur, il a donc les droits de tous. Quand un utilisateur crée un objet, il possède alors tous les droits de celui-ci. Le reste des utilisateurs n'ont pas de droits saufs si on lui attribue.

### Liste des privilèges

- **CREATE** : privilège permettant de créer de nouveaux objets
- **SELECT** : privilège permettant de lire le contenu d'une table ou vue.
- **INSERT** : privilège permettant d'insérer des valeurs.
- **UPDATE** : privilège permettant de modifier des valeurs.
- **DELETE** : privilège permettant d'effacer des tuples.
- **REFERENCES** : privilège permettant de faire référence à une table ou vue dans une contrainte d'intégrité.
- **ALL**: tous les privilèges.
- Autres privilèges : **RULE**, **TRIGGER**, **USAGE**, ...

Pour ajouter une permission à tout le monde, on utilise : **ALL** dans `[liste_users]`.

### GRANT

Pour ajouter des permissions, on utilise cette syntaxe :

```
GRANT [liste_privilèges] ON [liste_objets] TO [liste_users];
```

Exemple de commande :

```
GRANT SELECT,INSERT ON fournisseurs,produits TO pierre,paul;
```

On peut ajouter l'option **WITH GRANT OPTION** ce qui permet à l'utilisateur recevant la permission de, à son tour, donner cette même permission à un autre utilisateur. Exemple :

```
GRANT UPDATE ON commandes
TO anne WITH GRANT OPTION;
```

Donne les droits de mise à jour à Anne sur la table commandes avec possibilité de transmettre ce droit.

### REVOKE

Cette commande permet de retirer des permissions. On utilise cette syntaxe :

```
REVOKE [liste_privileges]
ON [liste_objets] FROM [liste_utilisateurs]
[CASCADE | RESTRICT];
```

L'option RESTRICT est par défaut si **CASCADE** n'est pas indiqué.

**CASCADE** n'affecte que les droits qui avaient été attribués à travers une chaîne d'utilisateurs traçable jusqu'à l'utilisateur qui subit la commande **REVOKE**. Donc, les utilisateurs affectés peuvent finalement garder le droit s'il avait aussi été attribué via d'autres utilisateurs.

Exemple de commande :

```
REVOKE UPDATE ON commandes FROM anne;
```

Retire les droits de mise à jour à Anne sur la table « commandes ».

## Les autres objets

### Les index

Les index ont pour unique objectif d'améliorer les recherches. Les index améliorent les recherches, mais pénalisent les mises à jour.

Pour créer un index, on exécute cette commande :

```
CREATE INDEX [nom_index]
ON ([colonne] [ordre de tri]);
```

On la supprime avec :

```
DROP INDEX [nom_index];
```

Il faut le réindexer régulièrement grâce à la commande :

```
REINDEX [nom_index];
```

## Les autres objets

Voici la liste des objets existants en SQL :

- (CREATE|ALTER|DROP) DATABASE
- (CREATE|ALTER|DROP) DOMAIN
- (CREATE|ALTER|DROP) FUNCTION
- (CREATE|ALTER|DROP) GROUP
- (CREATE|ALTER|DROP) LANGUAGE
- (CREATE|ALTER|DROP) INDEX
- (CREATE|ALTER|DROP) SCHEMA
- (CREATE|ALTER|DROP) SEQUENCE
- (CREATE|ALTER|DROP) TABLE
- (CREATE|ALTER|DROP) TRIGGER
- (CREATE|ALTER|DROP) USER
- (CREATE|ALTER|DROP) VIEW

Ils peuvent être créer, modifier et supprimer de la même manière d'une table.